HOW TO MAKE A FULLY AUTOMATED CD PIPELINE FOR AN ANGULAR FRONTEND APPLICATION

Amazon Route 53 | AWS CloudFormation | AWS CodeBuild | AWS CodeDeploy | AWS CodePipeline | AWS Lambda | CI/CD | Continuous Deployment | Continuous Integration | How-to

beSharp | 3 April 2020

We have already written about **Continuous Delivery/Continuous Integration** several times, most of them mainly focused on the backend part of an application. Of course, web applications also require a **front-end**, thus, making the deployment process of the front-end **agile and fast** is crucial for the success of a project.

In this article, we want to bring you the **complete project** for an AWS infrastructure to host an **Angular application**, complete with a CDN, a custom function to invalidate the cache and a fully **automated CD pipeline**.

This project aims to fulfill the best practices and will provide a robust, highly scalable and fully managed solution for the hosting of front-end applications on AWS. This is also the most **cost-optimized** solution we have found so far, allowing you to serve your static content in a very reliable, performant and cheap way.

Our architecture is logically partitioned in 2 parts, the customer-facing hosting, and the developer-facing Continuous Delivery infrastructure.

The services

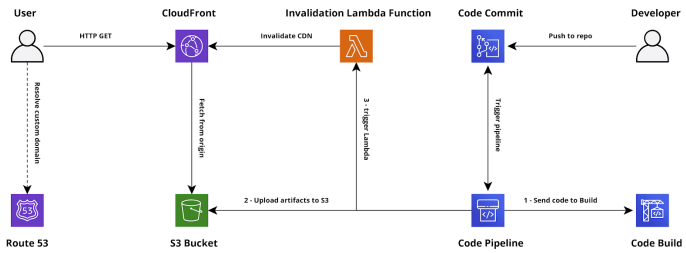
For the hosting, we will use the following AWS services:

- An **S3 Bucket** to store all the front-end files
- A **CloudFront Distribution** to serve the bucket content to final users
- **Route53** to configure and use a custom domain for the application

The developer-facing infrastructure will be made using:

- **CodeDeploy** as private Git repository
- **CodePipeline** as an **orchestrator** for the CD pipeline
- **CodeBuild** to provide the computing environment required to perform the angular build
- A custom **Lambda function** to create an invalidation for CloudFront at each deploy

The architecture



The diagram above shows the complete infrastructure, it also shows the triggers and the action taken during the pipeline execution.

The customer-facing hosting is based on CloudFront, which allows us to serve the Angular application in a very efficient and reliable way. Using CloudFront is also required to enable **HTTPS** for our front-end application.

The compiled assets of the front-end application will be stored inside an S3 Bucket, used as an origin for the CloudFront distribution. We strongly advise keeping the S3 Bucket private, in this way no external user will be able to directly access the files on S3, limiting costs and possibilities of exploits.

We included Route53 in the architecture because it's the most straightforward way to manage DNS records for the domain, anyway, any DNS service would do, as long as you can add a CNAME record for your front-end application.

The real interesting part of the architecture is the **developer-facing** one. The core of the solution is CodePipeline, which we use as an orchestrator to react to changes made in the repository and pass the data between all the services used during the pipeline execution.

We also used CodeCommit as a repository for the application code. At the time of writing, CodePipeline supports CodeCommit and GitHub. In order to support other Git repo configurations, **further customization** would be needed.

In order to build the angular solution, we leverage CodeBuild to automatically provision a container for the build process.

To speed up the deploy and avoid waiting for the expiration of each object in the CDN, we included an **Invoke step** at the end of the pipeline to run a Lambda Function which creates an invalidation request for the CloudFront Distribution.

The pipeline steps

Our pipeline will be composed of 4 steps, let's break down the pipeline mechanism.

Source

This step is fully managed by AWS CodePipeline. The step is configured to automatically start the pipeline when a push is detected in a specific branch, download the source code, make an archive and send it as an artifact for the next step.

Build

The build step needs the source code artifact from the previous step as an input.

CodeBuild will then provision a container for the build, download the source code and execute the commands contained in the buildspec.yml file.

The yml file must be stored in the root of the repository.

Here a sample **buildspec.yml** file for a standard Angular application:

```
version: '0.2'
phases:
  install:
    runtime-versions:
      nodejs: 12
  pre_build:
    commands:
      - echo "Prebuild, installing npm dependencies"
      - npm install
  build:
    commands:
      - echo "Starting the build step"
      - npm run build
      - echo "Finished"
artifacts:
  name: "BuildOutput"
  files:
    - '**/*'
  base-directory: 'dist'
```

The output of the build process is an archive of the files and folders contained in the dist folder.

Deploy

The deploy step is also completely managed, CodePipeline will unpack and copy to S3 all the files and folders of the build output archive.

At the time of writing, this step is not able to delete files from S3, usually, this is not a problem given the structure of a standard Angular application. However, keep in mind that if you want to delete a file from the website you have to add a step to empty the bucket before deploying or deleting it after the deploy with another step.

CloudFront Invalidation

The last step of the pipeline invokes a Lambda function to create an invalidation for the CloudFront distribution, allowing our users to obtain the updated version of the applications in few minutes after the deploy, instead of wait for objects expirations in each CDN node, which may occur at different times.

The lambda function can use the AWS SDK to create the invalidation. This function also needs to notify CodePipeline whenever it encounters errors or finishes with success using a specific CodePipeline API. Here an example lambda to completely invalidate a CloudFront distribution and then notify to CodePipeline the result:

```
import boto3
import os

from botocore.exceptions import ClientError

cloudfront_client = boto3.client('cloudfront')
codepipeline_client = boto3.client('codepipeline')


def lambda_handler(event, context):
    try:
        cdn_id = os.environ["CDN"]
        cloudfront_client.create_invalidation(
            DistributionId=cdn_id,
            InvalidationBatch={
                'Paths': {
                    'Quantity': 1,
                    'Items': [
                        '/*'
                    ],
                },
                'CallerReference': event['CodePipeline.job']['id']
            }
        )

        codepipeline_client.put_job_success_result(jobId=event['CodePipeline.job']
['id'])
    except ClientError as e:
        print("Boto3 exception", e)
        codepipeline_client.put_job_failure_result(
            jobId=event['CodePipeline.job']['id'],
            failureDetails={
                'type': 'JobFailed',
```

```
                'message': e.response
            })
    except Exception as e:
        print("Error", e)
        codepipeline_client.put_job_failure_result(
            jobId=event['CodePipeline.job']['id'],
            failureDetails={
                'type': 'JobFailed',
                'message': e.args
            })
```

How to build the solution

The following is a list of operations to follow in order to build the solution. Please note that this is not a copy and paste tutorial, it gives you the right order for each operation and an insight into what to do.

Before starting, make sure to have your source code in a CodeCommit repository and to have full access to the account to create and configure each service of the solution.

1. First, you need to create a private S3 Bucket for the front-end.
    1. Follow the wizard, be sure to select the options to make the bucket private.
    2. Choosing the bucket name as the FQDN is not required because we will serve the content using CloudFront
2. Create a CloudFront distribution.
    1. Select the Bucket created in the first step as the origin.
    2. Check the option to allow CloudFront to manage the bucket policy on your behalf, in this way the wizard will create a perfectly safe and valid policy for your bucket to allow the distribution to serve your content keeping the bucket private at the same time.
    3. Redirect all errors to the index page, this is crucial for Angular router to work.
    4. **NOTE:** If you are not creating the infrastructure in the us-east-1 region, please wait up to 2 hours before testing it. This is due to how the DNS propagation works in AWS at the time of this writing, a freshly created S3 bucket in any region different from us-east-1 will cause a temporary redirect that will break the s3 cloudfront integration.
3. Create the Lambda function to invalidate the CDN.
    1. You can copy and paste the code of the example above.
    2. If you use the example code, add an environment variable called “CDN” and set it with the CloudFront distribution id.
    3. Make sure to create a new IAM Role for the function, with all the permissions of the BasicLambdaExecution role, plus:
        1. cloudfront:CreateInvalidation on the CloudFront distribution
        2. codepipeline:PutJobSuccessResult on *
        3. codepipeline:PutJobFailureResult on *
    4. Take note of the Lambda function name and ARN for later.
4. Create a new CodePipeline

1. Follow the wizard and add the source step, select your CodeCommit repository and the desired git branch
2. Add a build stage, create a new CodeBuild project following the wizard
    1. Select a standard Ubuntu image
    2. Create or select a standard CodeBuild service role
    3. Leave all the defaults and continue
3. Add a deploy step
    1. Select S3 deploy
    2. Follow the wizard to configure it to push the build output on your front-end bucket
4. Finally, add the last step, choose Invoke -> AWS Lambda
    1. Fill out the form, and select the Lambda function created in the previous step
    2. Leave all the parameters on their default value
5. Complete the pipeline creation, and test it. The pipeline should start as soon as you hit the save button.

Congratulations! You completed the deploy of the solution! You should now have a fully functional Angular hosting, complete with a continuous delivery pipeline.

Still curious about **Continuous Delivery/Continuous Integration?**

Read also:

- How to create flexible CI/CD Pipeline on AWS with Fargate and SQS
- How to create and maintain an AWS serverless infrastructure with Troposphere and CodePipeline
- Pipelines di CD/CI cross-account con AWS CodePipeline (English coming soon!)

Leave a comment or contact us for questions or suggestions!

See you 🙂

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it